gpicamera

September 10, 2019
13:08

Contents

1 Routine to set up camera views for use with NSTX Gas Puff Imaging Experiment

This file serves as an example *usr2ddetector.web* routine for the more complicated case of a 2-D imaging camera. The particular parameters used here are from the 2004 NSTX run campaign. More recent versions differ only in the numerical values of these parameters; the basic principles remain the same.

A key distinction between this and the approach used in simulating one dimensional imaging arrays (e.g., as in *btopdetector.web*) is that memory considerations prohibit computing the simulated image as part of the primary simulation (i.e., *flighttest*). For example, in this case, the *de_zone_frags* array would have $64 \times 64 \times$ *zn_num* elements. For a typical 3-D GPI simulation, *zn_num* $\sim 10^5 \rightarrow 4 \times 10^8$ 64 bit real numbers or > 3 GB, quite a large amount of memory.

We instead compute a much lower resolution (represented by the *nx* and *ny* macro parameters below; both $= 5$) image during the run of *flighttest*; the resulting data are used only to check the simulation and camera geometry. The full camera image is computed subsequently via the *postdetector* routine. Because all of the constituent chord integrals are independent, each can be computed separately so that only *zn_num* elements are needed in *de_zone_frags* at a time. This independence also allows the calculation to be parallelized for efficiency. The particular subroutine called from *postdetector* is *not detector_setup*, but *gpi_views*. An important implication of this is that *postdetector* is presently specific to these GPI applications.

Each of the camera pixels corresponds to a single chord specified by two points, as is described in the *Defining Radiation Detectors* section of the User's Manual. One of these points is the camera vertex, represented by the macros *x_p0*, *y_p0*, *z_p0* below. The "p" in the macro refers to the fact that the DEGAS 2 values for these coordinates are in a system rotated toroidally from the experimental coordinate system; this is explained further below.

The second point of each chord can be determined by whatever means are available. In all GPI simulations to date, these second points are obtained as a mapping between the camera's pixel coordinates and physical coordinates on a "target plane". Conceptually, this is just a plane centered on the gas puff. In the NSTX case, it is defined in principle by a line (the gas manifold, a stainless steel tube) and a vector pointing at the center stack. In reality, it corresponds to a rectangular piece of metal bolted to the gas manifold.

Coordinates of arbitrarily located physical points on this "plane" are taken via a precision measuring arm, and a plane of the form

$$a_0 x + a_1 y + a_2 z = 1$$

is fit to the resulting data. For convenience, the DEGAS 2 simulations place the center of the gas manifold at zero toroidal angle. Since the measurement arm data are referenced to a different location, a toroidal rotation *phi* must be applied to the target plane before use here. The end result is a second set of plane coefficients, $a_0' \rightarrow$ *a0_p*, etc. As noted above, this same rotation is applied to the coordinates of the camera vertex.

The gas puff for GPI experiments on Alcator C-Mod is provided by one or more capillaries at a single toroidal angle. In this case, the target plane is simply a vertical plane at that angle. The adaptation of this file to those experiments would require changing the values of the plane coefficients. Examples of such specifications can be found in *postdetector.web*, where the same target plane data are also used.

The mapping of points on this target plane to pixel coordinates is determined from an image of this same calibration target plane recorded by the camera (in NSTX, the process is rendered more complex by the optical system, although the overall procedure is the same). The particular approach used in NSTX consists first of using the resulting image to fit the mapping between the pixel coordinates (c_x, c_y) and spatial coordinates (R, Z):

$$R = a_{Rx}c_x + a_{Ry}c_y + R_0$$

$$Z = a_{Zx}c_x + a_{Zy}c_y + Z_0,$$

where c_x ranges from *cx_min* to *cx_max*, and c_y ranges from *cy_min* to *cy_max*. Note that the coefficients in these equations do not appear below as variables or macros but simply as constants in the expressions for *r_maj* and *z*. Also note that relative spatial calibrations made during the campaign indicated that the camera view had shifted in the c_y direction, leading to the shift in that coordinate noted below.

But, in a 3-D simulation we need to know all three coordinates, (x, y, z) in the target plane for each pixel. Obviously, $z(c_x, c_y) = Z(c_x, c_y)$. The values of $x(c_x, c_y)$ and $y(c_x, c_y)$ are obtained from a quadratic equation derived by combining the definition of $R = \sqrt{x^2 + y^2}$ with the above equation for the target plane.

`$Id: e5cf18133abfe04fa21a6491e8eb68b064c8a7fd $`

"gpicamera.f" 1 ≡
@m FILE 'gpicamera.web'

The unnamed module.

"gpicamera.f" 1.1 ≡
⟨ Functions and Subroutines 1.2 ⟩

GPI camera views.

```
"gpicamera.f" 1.2 ≡
  @m nx 5    // Number of chords in radial direction.
  @m ny 5    // Number of chords in poloidal direction.
  @m cx_min zero
  @m cx_max const(6.3, 1)    // Number of radial pixels in real camera.
  @m cy_min zero
  @m cy_max const(6.3, 1)    // Number of poloidal pixels in real camera.
  @m a0 const(−3.60752)    // Coefficients of the camera"s "target plane"
  @m a1 const(2.11865)
  @m a2 const(3.17544)
  @m phi const(6.146, 1) ∗ PI / const(1.8, 2)    // Toroidal shift to my system
  @m x_p0 const(1.61388)    // Camera vertex
  @m y_p0 const(−0.53975)
  @m z_p0 const(−0.214046)

⟨ Functions and Subroutines 1.2 ⟩ ≡
    subroutine detector_setup

      implicit_none_f77
      de_common
      zn_common
      implicit_none_f90

      ⟨ Memory allocation interface 0 ⟩

      detector_total_views = nx ∗ ny
      var_alloc(de_view_points)
      var_alloc(de_view_algorithm)
      var_alloc(de_view_halfwidth)

      call initialize_zone_frags

      de_grps = 0
      de_view_size = 0
      var_alloc(de_view_tab)

      call detector_setup_a

      return
    end
```

See also sections 1.3 and 1.4.

This code is used in section 1.1.

Extension of the above subroutine. Statements actually making assignments to the detector pointer arrays (*de_zone_frags* specfically) need to be separated from their allocation above so that their array indexing gets handled correctly.

```
⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine detector_setup_a

    define_varp(zone_frags, FLOAT, zone_ind)

    implicit_none_f77
    de_common
    zn_common
    implicit_none_f90

    integer view, num, var, tab_index, spacing, ix, iy, i, zone, ix_max, iy_max
    integer grp_views_{nx*ny}
    real var_min, var_max, mult

    declare_varp(zone_frags)

    ⟨ Memory allocation interface 0 ⟩

    var_alloc(zone_frags)

      /* Use local variables for nx and ny so the arguments to gpi_views can have other values than
         those given by the macros at the top of this file. */
    ix_max = nx
    iy_max = ny
    do ix = 1, nx
      do iy = 1, ny
        view = (iy − 1) * nx + ix
        call gpi_views(ix, ix_max, iy, iy_max, vc_args(de_view_points_{view,de_view_start}),
            de_view_halfwidth_{view}, de_view_algorithm_{view}, zone_frags)    /* This routine inserts
            zone_frags into the common array de_zone_frags, compressing the data in the porcess. */
        call add_zone_frags(view, zone_frags)

      end do
    end do

    var_free(zone_frags)

    num = nx * ny
    var = de_var_unknown
    tab_index = zero
    var_min = zero
    var_max = zero
    mult = zero
    spacing = de_spacing_unknown
    do i = 1, num
      grp_views_i = i
    end do
    call de_grp_init('GPI␣chords', num, var, tab_index, var_min, var_max, mult, spacing, grp_views)

    return
  end
```

Details of the GPI camera views. These are separated here so that they can be called by a post-processing routine, separate from the detector class arrays.

```
⟨Functions and Subroutines 1.2⟩ +≡
    subroutine gpi_views(ix, ix_max, iy, iy_max, vc_dummy(points), halfwidth, algorithm, zone_frags)
      implicit_none_f77
      zn_common
      implicit_none_f90

      integer ix, ix_max, iy, iy_max     // Input

      integer algorithm     // Output
      real halfwidth

      vc_decl(points_{de_view_start:de_view_end})
      real zone_frags_{zn_num}

      real delta_cx, delta_cy, a0_p, a1_p, a2_p, cx, cy,     /* Local */
        r_maj, z, disc, x_p, y_p, z_p

      assert(ix_max > 1)
      assert(iy_max > 1)
      delta_cx = (cx_max − cx_min) / areal(ix_max − 1)
      delta_cy = (cy_max − cy_min) / areal(iy_max − 1)
           /* Rotate Ricky"s "target plane" to my coordinate system. The "p" denotes "prime". */
      a0_p = a0 * cos(phi) + a1 * sin(phi)
      a1_p = a1 * cos(phi) − a0 * sin(phi)
      a2_p = a2
      cx = cx_min + areal(ix − 1) * delta_cx
      cy = cy_min + areal(iy − 1) * delta_cy
           /* This was the original mapping provided by Ricky Maqueda. */
  @#if 0
      r_maj = const(1., −3) * (−const(0.7555) * cx − const(3.5159) * (cy + const(2.67)) + const(1.604, 3))
      z = const(1., −3) * (const(2.8476) * cx − const(0.5254) * (cy + const(2.67)) + const(1.31, 2))
  @#endif
           /* Subsequently concluded that a 6 pixel was appropriate for these shots: */
      r_maj = const(1., −3) * (−const(0.7555) * cx − const(3.5159) * (cy + const(6.0)) + const(1.604, 3))
      z = const(1., −3) * (const(2.8476) * cx − const(0.5254) * (cy + const(6.0)) + const(1.31, 2))
         /* Solve for corresponding Cartesian coordinates in my system using the rotated target plane and
            R^2 = x^2 + y^2. */
      z_p = z
      disc = (a0_p^2 + a1_p^2) * r_maj^2 − (one − a2_p * z_p)^2
      assert(disc > zero)
      x_p = (a0_p * (one − a2_p * z_p) + a1_p * sqrt(disc)) / (a0_p^2 + a1_p^2)
      y_p = (a1_p * (one − a2_p * z_p) − a0_p * sqrt(disc)) / (a0_p^2 + a1_p^2)     /* The "4" here accounts
            for the fact that there are two computed variables, x_p and y_p, and that they are squared. */
      assert(abs(r_maj^2 − x_p^2 − y_p^2) < const(4.0) * epsilon)
      algorithm = de_algorithm_circular
      halfwidth = const(3.9, −1) * PI / const(1.8, 2)
      vc_set(points_{de_view_start}, x_p0, y_p0, z_p0)
      vc_set(points_{de_view_end}, x_p, y_p, z_p)
      call detector_view_setup(vc_args(points_{de_view_start}), halfwidth, algorithm, zone_frags)

      return
    end
```

2 INDEX

⟨ Functions and Subroutines 1.2, 1.3, 1.4 ⟩ Used in section 1.1.
⟨ Memory allocation interface 0 ⟩ Used in sections 1.3 and 1.2.

COMMAND LINE: `"fweave -f -i! -W[ -ybs15000 -ykw800 -ytw40000 -j -n/ /Users/dstotler/degas2/src/gpicamera.web"`.

WEB FILE: `"/Users/dstotler/degas2/src/gpicamera.web"`.

CHANGE FILE: `(none)`.

GLOBAL LANGUAGE: FORTRAN.